A New timed Petri net model for loop scheduling with resource constraints

Jian Wang, Christine Eisenbeis

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1810

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

A NEW TIMED PETRI NET MODEL FOR LOOP SCHEDULING WITH RESOURCE CONSTRAINTS

Jian Wang
Christine Eisenbeis

Décembre 1992

A NEW TIMED PETRI NET MODEL FOR LOOP SCHEDULING WITH RESOURCE CONSTRAINTS *

ORDONNANCEMENT DE BOUCLES AVEC CONTRAINTES DE RESSOURCES: UNE NOUVELLE MODÉLISATION PAR RÉSEAU DE PETRI TEMPORISÉ

Jian Wang †
Christine Eisenbeis ‡

INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

*This work was partially supported by ESPRIT Project COMPARE
†e-mail: Jian.Wang@inria.fr; Tel:33-1-39635361; Fax: 33-1-39635330
‡e-mail: Christine.Eisenbeis@inria.fr

Abstract

This report uses timed Petri net to model and analyze the problem of instruction-level loop scheduling with resource constraints, which has been proven to be an NP complete problem. First, we present a new timed Petri net model to integrate loop scheduling, functional unit allocation, register allocation and spilling into a unified theoretical framework. Secondly, we theoretically discuss the schedulability of our model. Thirdly, we develop a state subgraph, called Register Allocation Solution Graph, which can effectively describe the major behavior of our new model. The main property of this state subgraph is that the number of all its nodes is polynomial. Finally we present a timed Petri net based loop scheduling approach with polynomial computation complexity, by which the optimum loop schedules can be found under resource constraints for almost all practical loop programs.

Résumé

Dans ce rapport, on utilise un réseau de Petri temporisé pour modéliser et analyser le problème de l'ordonnancement des boucles à bas niveau, avec contraintes de ressources, problème reconnu NP-complet. D'abord, nous construisons la modélisation par réseau de Petri, qui intègre à la fois l'ordonnancement des instructions, l'allocation des unités fonctionnelles, l'allocation de registres et le spilling en mémoire. Nous discutons de la réalisabilité d'un ordonnancement basé sur ce modèle. Puis nous proposons un sous-graphe des états, appelé Graphe de Résolution de l'Allocation de Registres, qui décrit des solutions dominantes de notre problème, et dont la propriété principale est que le nombre de ses nœuds est polynomial. Enfin, sur cette base, nous présentons une approche à l'ordonnancement de boucles, de complexité polynomiale, qui permet de trouver les ordonnancements optimaux sous contraintes de ressources, sous des conditions vérifiées par la plupart des boucles dans les programmes.

1 Introduction

It is now widely thought that future high performance computer architectures will not only profit from (coarse grain) parallelism between processors, but also from improvement of one single processor, mainly due to two kinds of (fine grain) parallelism: pipelining and duplication of functional units (pipelined, VLIW, superscalar processors). Compilers are therefore expected to be able to take advantage of offered parallelism, by extracting and exploiting instruction level parallelism as much as possible. Loop optimization is a major challenge in this domain, since loops represent the most consuming execution time in scientific codes, among which scientific problems identified as the "Grand Challenges" of next years.

Fine grain parallelism optimization can be modelled as a scheduling problem with resource constraints, known as an NP-complete problem. That is why much of the work in compiler optimization has focused to development and evaluation of empirical heuristics, often targeted to specific architectures. However, loop scheduling is a subproblem of general scheduling problem, where the criteria to optimize is the throughput of iterations per unit time since the number of iterations is often supposed to be great enough. In this context, some polynomial cases have been theoretically identified, and heuristics precisely evaluated. But this could be done only at the price of simplifications (resource constraints not taken in consideration, register allocation treated apart of operations scheduling).

In this report, we present the first model, based on a Timed Petri Net (TPN), of whole loop scheduling problem. Not only the problems of functional units and register allocation are addressed, but also the possibility of spilling variables into memory, when no more register is available. To our knowledge, this is the first Petri net model of spilling.

More, we show that solutions to loop throughput optimization problem can be characterized by their effect on only a subset of places. Therefore, performance analysis is done through a subgraph of the state graph (called RASG for Register Allocation State Graph). This permits to derive the following important result: for a subclass of loops, verifying a certain condition (to be described in section 4, in fact, most loops verify this condition) and a given architecture (fixed number of registers), the problem of loop optimization with spilling is polynomial.

Our report is organized as follows. In section 2, we review previous works in loop optimization, and especially previous models by timed Petri nets. Section 3 describes the construction of our timed Petri net model from loop optimization specifications and focuses on spilling model. In section 4 we theoretically discuss the schedulability of our new model. In section 5, we explain how to build the Register Allocation State Graph, and we describe how to use it for finding an optimal loop schedule in presence of resources constraints. In section 6 conditions on loops for polynomiality are discussed and loop transformations are proposed for achieving these conditions. We conclude this report in section 7.

2 Background

Since loops constitute the most consuming part of practical programs, their optimization is a crucial key in code optimization. Due to the increasing low-level parallelism offered by modern computer architectures, optimizing operation scheduling within one iteration is often not sufficient and operations from different iterations are required to run in parallel.

The problem can be stated as follows: we consider a set of operations (the loop body) to be run on a processor. Processor architecture is given by its functional units (memory access, integer unit, floating point unit, ...) and register files. The optimization problem is to find a scheduling of operations such that

Semantic Constraints The order of operations is such that data dependencies are verified. Data dependencies are given under the form of a Loop Data Dependence Graph (LDDG) where the nodes are the operations and an edge between two nodes means that a data dependence exists between two instantiations of this operation.

Resource Constraints Two operations sharing the same functional unit can not be issued at the same time and two intermediate values simultaneously alive can not be allocated in the same register. Actually, [Eis89] proves that, modulo a possible loop unwinding, the latter constraint amounts to: at each time, there are no more simultaneously alive variables that the number of available registers.

Optimization criteria The loop throughput is maximum: loop throughput is defined as the average number of operations performed per time unit.

Beside simple techniques such as loop unrolling in view of increasing the number of operations in one single iteration, more sophisticated optimization techniques are based on a rewriting of the loop, consisting of operations from different iterations. The generic term for such a transformation is "software pipelining". Methods for software pipelining are based either on the determination of a repetitive pattern by unwinding the loop iteration by iteration and scheduling the code [Aik87], or the direct construction of the pattern that forms the body of transformed loop [Cha81,Tou84,Eis88,Lam88,Bod89,Su91]. In this context, "resource constraints" refers only to processors or functional units allocation. The register allocation problem is treated apart or not addressed.

2.1 Instruction-Level Loop Scheduling with General Resource Constraints

In presence of general resource constraints, loop low level code optimization consists in determining at compile time not only operations scheduling, but also resource allocation (especially register allocation). These problems are closely related since, for instance, allocating the same register to two different variables makes the simultaneous execution of corresponding operations impossible.

Up to now, operation scheduling and register allocation were treated apart, and only the problem of performing register allocation before or after scheduling was raised. Most authors conclude that register allocation should be done after scheduling, see [Bra91], whose studies are done in the context of linear code (basic blocks scheduling). Same conclusion appears from experiments in loop code scheduling [Lam88,Eis89]. But the question of what to do when there are not enough registers available remains: spilling operations must be introduced, but they may disturb and damage the schedule. In that case, [Lam88] gives up software pipelined code and returns to simple loop iteration scheduling. [Eis89] uses a heuristic method to spill variables into local memory (which can be accessed in parallel with other functional units). Therefore previous work presents no global view about whole problem of loop scheduling and register allocation, by taking spilling in consideration.

2.2 The Existing TPN Models for Loop Scheduling

Two previous works [Han87,Gao91] have already considered a Petri Net model of loop scheduling in presence of resource constraints. In these models, loop operations are modelled by transitions, whereas resources are modelled by places. Edges represent either dependence constraints or use/releasing of resources.

Main contribution of [Han87] is a very precise model of functional units, by considering in details their working. For instance, every stage of a pipeline unit is considered as a resource and tasks running on these units are splited into subtasks. This model permits to take into account also complex architectures such as microprogrammed processors. Duplicated functional units and more than one register file can be modelled, but only if they can be interchanged. Else the choice of unit for running an operation can not be modelled and must be specified. One minor drawback of this model is that simultaneous reading/writing into a register can not be specified. Another important restriction is the hypothesis of "non-re-entrance" tasks, i.e. two successive iterations of the same operation can not overlap. This explains why pipelines can not be modelled by considering one single transition. The complexity of resulting State Graph is exponential in the number of input tasks of the loop body.

Unlike Hanen's work, [Gao91] considers more simple architectures, mainly data flow architectures, i.e. essentially pipelined operations and buffering of intermediate results. In their first model [Gao91], they consider the problem without resource constraints, find that a periodic behavior is observed under the rule that a transition is fired as early as possible and give polynomial bounds for this periodic behavior to occur (for transitions on the critical cycles). By exploiting this model further [Gao92], they establish that minimizing the amount of storage required for achieving optimal rate for loops is a polynomial problem for loops without loop carried dependencies. The problem of what to do when there is not enough registers in the architecture remains.

Our work is a generalization of latter work, first because we are able to model the spilling process of variables and therefore we obtain the optimal loop rate for the given architecture, second because our model can handle a larger class of loops (the restriction given in section 4 is verified by most practical programs).

2.3 Some Useful Notations on Petri Net and Timed Petri Net

This subsection gives some notations on Petri net [Com71,Pet81,Rei85] and timed Petri net [Car84,Car88] which will be used throughout this paper.

Definition 2.1 A Petri net is a five-tuple (P, T, A, W, M), where P is a non-empty set of places, T is a non-empty set of transitions and A is a non-empty set of directed arcs from transitions to places or from places to transitions. W, called as weight, is a function from A to the non-negative integers N, $W : A \rightarrow N$. M, called as marking, is a function from P to N, $M : P \rightarrow N$. We can use a graph to represent a Petri net in which P, T and A are pictorially represented by circles, bars and directed arcs, respectively. We also denote

$$I(p) = \{t \mid (t,p) \in A\}, \forall p \in P;$$
$$O(p) = \{t \mid (p,t) \in A\}, \forall p \in P;$$
$$I(t) = \{p \mid (p,t) \in A\}, \forall t \in T;$$
$$O(t) = \{p \mid (t,p) \in A\}, \forall t \in T;$$

where (t,p) denotes the directed arc from t to p while (p,t) the directed arc from p to t.

Definition 2.2 A transition $t_j \in T$ in a Petri net (P, T, A, W, M) is enabled if for all $p_i \in P$

$$M(p_i) \geq \#(p_i, I(t_j))$$

where

$$\#(p_i, I(t_j)) = \begin{cases} W((p_i, t_j)) & \text{if } p_i \in I(t_j) \\ 0 & \text{if } p_i \notin I(t_j) \end{cases}$$

Definition 2.3 A transition t_j in a Petri net (P, T, A, W, M) may fire whenever it is enabled. Firing an enabled transition t_j results in a new marking M' defined by

$$M'(p_i) = M(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

for all $p_i \in P$, where

$$\#(p_i, O(t_j)) = \begin{cases} W((t_j, p_i)) & \text{if } p_i \in O(t_j) \\ 0 & \text{if } p_i \notin O(t_j) \end{cases}$$

Definition 2.4 [Rei85] Let PN be a Petri net, $PN = (P, T, A, W, M)$, and let $P_s \subseteq P$.

(1) P_s is called a deadlock if and only if $\#(P_s, I(t_j)) \geq \#(P_s, O(t_j))$ for every $t_j \in T$.

(2) P_s is called a trap if and only if $\#(P_s, I(t_j)) \leq \#(P_s, O(t_j))$ for every $t_j \in T$.

(3) P_s is called a deadlock&trap if and only if $\#(P_s, I(t_j)) = \#(P_s, O(t_j))$ for every $t_j \in T$. where

$$\#(P_s, I(t_j)) = \sum_{\forall p_i \in P_s} \#(p_i, I(t_j)), \quad \#(P_s, O(t_j)) = \sum_{\forall p_i \in P_s} \#(p_i, O(t_j))$$

Definition 2.5 A Petri net (P, T, A, W, M) is a marked graph if and only if $\forall p_i \in P, |I(p_i)| = |O(p_i)| = 1$.

Definition 2.6 A timed Petri net is a six-tuple (P, T, A, W, M, D), where (P, T, A, W, M) is a Petri net and D is a function from the set of transitions to the nonnegative integers N, $D : T \rightarrow N$. $D(t)$ denotes the execution time (or the firing time) taken by transition t.

Definition 2.7 Let M_u denote the marking at time u, at time u, firing an enabled transition t_j results in a new marking $M'_{u+D(t_j)}$ at time $(u + D(t_j))$ and a new marking M'_u at time u, these two marking are defined by

$$M'_u(p_i) = M_u(p_i) - \#(p_i, I(t_j))$$

$$M'_{u+D(t_j)}(p_i) = M'_u(p_i) + \#(p_i, O(t_j))$$

for all $p_i \in P$.

Theorem 2.1 If P_s is a deadlock&trap then, for any marking M and any firable sequence of transitions, the number of tokens in P_s is not changed.

Theorem 2.2 A marking is live for a marked graph, MG, if and only if the number of tokens of each simple cycle in MG is positive.

Theorem 2.3 For a marked graph, a marking which is live remains live after firing.

3 A Modified TPN Model for Instruction-Level Loop Scheduling with Resource Constraints

In this section we modify Carlier's timed Petri net [Car84, Car88] to model instruction-level loop scheduling with resource constraints, in which loop scheduling, functional unit allocation, register allocation and spilling can be integrated into a unified theoretical framework.

The existing TPN models for loop scheduling [Han87, Gao91] are directly based upon Carlier's timed Petri net, in which time is introduced by such a manner that an execution time $d(t_i)$ is associated with transition t_i and the tokens are added to each output place of t_i only after t_i finishes its execution. However, this manner cannot reflect the fact in practical computers that a data can be read from a register and a new data can be written into the same register in one machine cycle and, also, different pipelined operations may exist in an instruction.

Hence, as the first step, we modify Carlier's timed Petri net and present General Modified timed Petri net (denoted as GMTPN) in which a time is put on each arc between place and transition, instead of on each transition.

Definition 3.1 A GMTPN is a six-tuple (P, T, A, W, M, D), where (P, T, A, W, M) is a Petri net and D is a function from A to the nonnegative integers $N, D : A \rightarrow N$.

Definition 3.2 Let M_u denotes the marking of a GMTPN at time u. At time u, firing an enabled transition t_j results in the following new markings:

for all $p_i \in P$

if $p_i \notin I(t_j) \cap O(t_j)$

$$M'_{u+D((p_i,t_j))}(p_i) = M_u(p_i) - \#(p_i, I(t_j))$$

$$M'_{u+D((t_j,p_i))}(p_i) = M_u(p_i) + \#(p_i, O(t_j))$$

if $p_i \in I(t_j) \cap O(t_j)$ and $D((p_i, t_j)) < D((t_j, p_i))$

$$M'_{u+D((p_i,t_j))}(p_i) = M_u(p_i) - \#(p_i, I(t_j))$$

$$M'_{u+D((t_j,p_i))}(p_i) = M'_{u+D((p_i,t_j))}(p_i) + \#(p_i, O(t_j))$$

if $p_i \in I(t_j) \cap O(t_j)$ and $D((p_i, t_j)) \geq D((t_j, p_i))$

$$M'_{u+D((t_j,p_i))}(p_i) = Mu(p_i) + \#(p_i, O(t_j))$$

$$M'_{u+D((p_i,t_j))}(p_i) = M'_{u+D((t_j,p_i))}(p_i) - \#(p_i, I(t_j))$$

We will point out that our modified TPN model is a subclass of GMTPN.

Next we notice that, in the existing TPN models for loop scheduling, register spilling has not been modelled which is a key of instruction-level loop scheduling with resource constraints. The model in [Gao91] doesn't consider register allocation and spilling, and the one in [Han87] models register allocation but no register spilling. In order to model register allocation and register spilling in a unified way, we introduce a variable place for each variable needed to be allocated in registers and the number of tokens in a variable place denotes the number of registers allocated to its corresponding variable. We also introduce store-spill transition from variable place to register place and load-spill transition from register place to variable place so the spilling can be modeled.

Now we can define our modified TPN (denoted as MTPN) model for instruction-level loop scheduling with resource constraints.

Definition 3.3 MTPN is a subclass of GMTPN. In terms of the LDDG of the given loop program and the given computer architecture, MTPN can be constructed by the following steps. For each operation *op* of LDDG, let $Def(op)$ and $Use(op)$ be the sets of variables defined and referenced by *op*, respectively.

1. Construct a marked graph from LDDG: For each operation of LDDG, build a transition

(called operation transition); for each edge $e = (op_i, op_j)$ of LDDG, build a place (called operation place) p and two arcs, from the operation transition corresponding to op_i (denoted as $t(op_i)$) to p and from p to $t(op_j)$. Let P_o represent the set of operation places, T_o the set of operation transitions and A_o the set of arcs.

2. Build register place and variable places: (1) Build a place (called register place) denoted as p_r; (2) for each variable x defined by operation of LDDG, build a place (called variable place) $p_v(x)$; (3) if operation op defined variables, then build an arc from p_r to $t(op)$; (4) if op defines x then build an arc from $t(op)$ to $p_v(x)$; (5) for each operation op referencing x, build an arc from $p_v(x)$ to $t(op)$. Let P_v represent the set of variable places.

3. Build releasing places and releasing transitions: (1) For each variable place $p_v(x)$ corresponding to variable x, build a transition (called releasing transition) $t_{rl}(x)$ and two arcs, from $p_v(x)$ to $t_{rl}(x)$ and from $t_{rl}(x)$ to p_r; (2) for each operation op referencing x, build a place (called releasing place) $p_{rl}(x, op)$ and two arcs, from $t(op)$ to $p_{rl}(x, op)$ and from $p_{rl}(x, op)$ to $t_{rl}(x)$. Let P_{rl} represent the set of releasing places and T_{rl} the set of releasing transitions.

4. Build spilling transitions: For each variable place $p_v(x)$, build two transitions (called spilling transition), store-spilling(x) and load-spilling(x) (denoted as $t_{ss}(x)$ and $t_{ls}(x)$, respectively), and build four arcs, from $p_v(x)$ to $t_{ss}(x)$, from $t_{ss}(x)$ to p_r, from p_r to $t_{ls}(x)$ and from $t_{ls}(x)$ to $p_v(x)$, respectively. Let T_{s-l} represent the set of spilling transitions.

5. Build functional unit places: For each functional unit fu, build a place (called functional unit place), $p_s(fu)$; for each operation op using fu, build two arcs, from $p_s(fu)$ to $t(op)$ and from $t(op)$ to $p_s(fu)$. Let P_s represent the set of functional unit places.

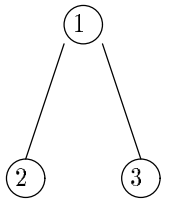
6. The definition of weight: For each arc e from p_r to $t(op)$, $W(e) = |Def(op)|$; for each arc e from variable place to operation transition, $W(e) = 0$; for other arc e, $W(e) = 1$.

7. The definition of time: (1) For each arc e from operation transition $t(op)$ to operation place, $D(e) = \delta(e')$, e' is the edge of LDDG corresponding to the operation place; (2) for each arc e from load-spill transition to variable place, $D(e) =$ the execution time of load operation; (3) for each arc e from operation transition $t(op)$ to functional unit place $p_s(fu)$, if fu is pipelined unit, then $D(e) = 1$; else $D(e) =$ the execution time of op; (4) for other arc e, $D(e) = 0$.

Provided the loop program and the computer architecture are given, the initial marking of the MTPN can be defined as follows.

Definition 3.4 Let M_0 be the initial marking of the MTPN, then
(1) $M_0(p_r)$= the number of registers;
(2) $M_0(p_i) = 0,\ \forall p_i \in P_{rl} \cup P_v$;
(3) $M_0(p_i) = \lambda(I(p_i), O(p_i)),\ \forall p_i \in P_o$;
(4) $M_0(p_j)$=the number of functional units corresponding to $p_j,\ \forall p_j \in P_s$.

Fig.3.1 gives an example of MTPN in which the LDDG of a loop is given in Fig.3.1(a) and we assume that there are two functional units and three registers in the computer architecture.

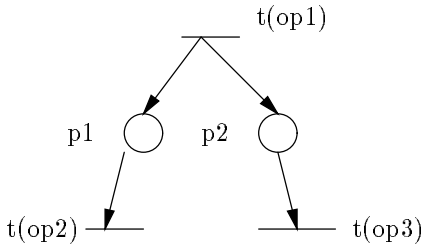


Def(op1)=x

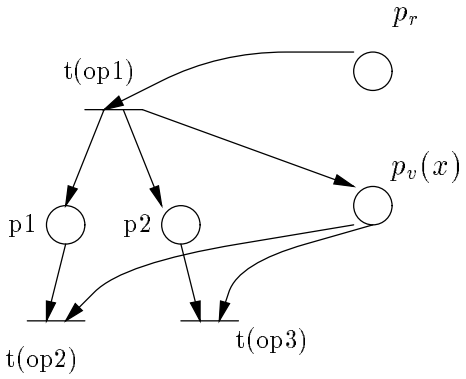
Use(op2)=x

Use(op3)=x

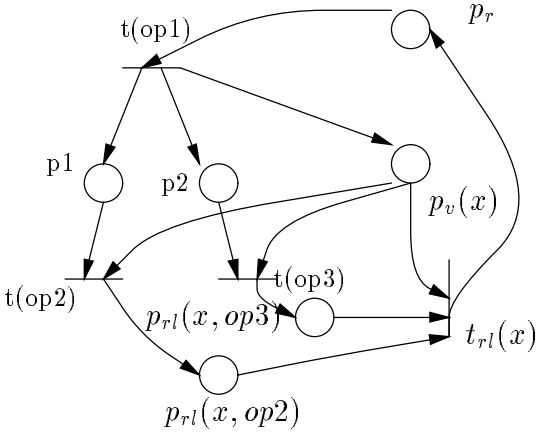
(a) LDDG



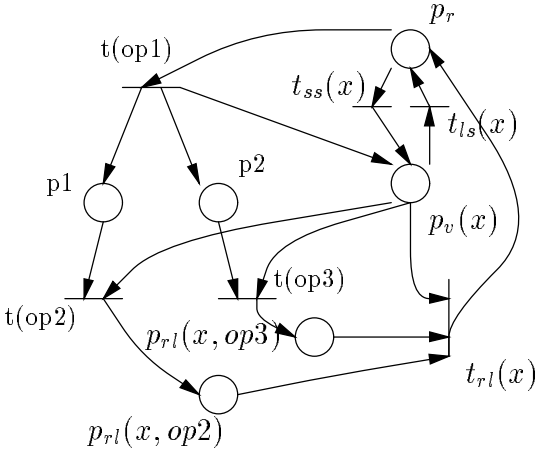
(b) A marked graph (Po,To,Ao)



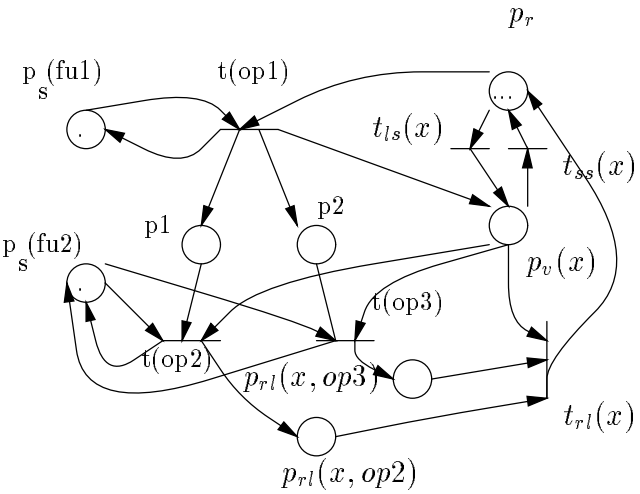
(c) After step 2



(d) After step 3



(e) After step 4



(f) After step 5

Fig.3.1 An example of MTPN

4 Schedulability of the MTPN

In this section, we will theoretically discuss the schedulability of a MTPN with an initial marking.

Definition 4.1 A schedule S of a $MTPN$ with an initial marking M_0 is a function from the set of nonnegative integers to the set of transitions and satisfies
(1) at any time u, $M_u(p_i) \geq 0$, for each $p_i \in P_o \cup P_{rl} \cup P_s \cup \{p_r\} \cup P_v$;
(2) for any pair of transitions, $(t_i^{(k)}, t_j^{(l)})$, in S, assume $t_i^{(k)}$ is scheduled at time u_i and $t_j^{(l)}$ at u_j, if $O(t_i) \cap I(t_j) \neq \emptyset$ *and* $l - k = M_0(O(t_i) \cap I(t_j))$ *and* $t_i, t_j \in T_o$, then

$$u_j - u_i \geq D(t_i, O(t_i) \cap I(t_j))$$

where $t_i^{(k)}$ denotes k^{th} firing of t_i, and $t_j^{(l)}$ l^{th} firing of t_j.

Theorem 4.1 (Model validity) Let $MTPN = (P, T, A, W, M_0, D)$, if σ is a firable sequence of its underlying Petri net (referred to as UPN-MTPN), (P, T, A, W, M_0), there exists a schedule for σ.

Proof: Let $S' = \sigma$, if a pair of transitions in S' doesn't satisfy the timing constraint, we can insert null transitions between these two transitions such that the timing constraint between them is satisfied. This work can be done for any pair of transitions in S', so the resulting sequence of S' will satisfy the constraint (2) of Definition 4.1. The constraint (1) is obviously satisfied, too.
□

Definition 4.2 A N-schedule of a $MTPN$ with an initial marking M_0 is a schedule in which each transition in T_o is fired N times.

Definition 4.3 Let σ be a firable sequence of an $UPN - MTPN$, and let $\bar{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_n)$ which is the vector of the firing number of each transition in T_o, $n = |T_o|$. Obviously, if σ corresponds to a N-schedule, then $\bar{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_n)$, where $\sigma_i = N$, $i = 1, 2, ..., n$.

Definition 4.4 A MTPN with an initial marking is schedulable if there exists an 1-schedule.

Lemma 4.1 Let $(UPN - MTPN, M_0)$ be an UPN-MTPN with initial marking M_0, if M_0 is live for each transition of T_o, then M_0 is live for $UPN - MTPN$.

This lemma is very obvious from Definition 3.3.

Lemma 4.2 Let $(UPN - MTPN, M_0)$ be an UPN-MTPN with an initial marking M_0, then $M_0(P_o)$ is a live marking for (P_o, T_o, A_o).

Proof: Note that (P_o, T_o, A_o) is a marked graph from Definition 3.3, and by Definition

3.4 and Theorem 2.2, $M_0(P_o)$ is live for (P_o, T_o, A_o).
□

Theorem 4.2 For any UPN-MTPN, $UPN-MTPN$, there exists a nonnegative integer K, if an initial marking M_0 satisfies that the number of registers, $M_0(p_r)$, is not less than K, then M_0 is live for $UPN - MTPN$.

Proof: By Lemma 4.1 and 4.2, we can only consider T_o and register place p_r.

Let $K = \#(T_o, O(p_r))$. If we let $M_0(p_r)$ be K, then the number of tokens of p_r is not less than $\#(T_o, O(p_r))$, it means p_r has enough tokens to fire any transition of T_o. Also, from the definition of MTPN, for each transition t_i of T_o, if $I(t_i)$ includes places of P_v, then there must be some transitions of T_o on which t_i is dependent. When these transitions are fired, they must put tokens into the places of P_v that $I(t_i)$ includes, so these places have enough tokens to fire t_i.
□

We are more interested in the minimum of K, so we give an algorithm to compute it as follow.

Algorithm 4.1

1. For each transition t_i of T_o
   1.1 compute $\#(t_i, O(p_r))$;
   1.2 compute $|I_{P_v}(t_i)|$, where $I_{P_v}(t_i) = \{p|(p, t_i) \in \bar{A}_o \; and \; p \in P_v\}$;
   1.3 let$RN(t_i) = \#(t_i, O(p_r)) \; + \; |I_{P_v}(t_i)|$;

2. For each place p_i of P_v
   2.1 find $S_{T_o}(p_i) = \{t_j | t_j \in T_o \; and \; t_j \in O(p_i)\}$;
   2.2 for any pair of $(t^{'}, t^{''})$ in $S_{T_o}(p_i)$, if $t^{'}$ is dependent on $t^{''}$, then remove $t^{''}$ from $S_{T_o}(p_i)$. So we obtain a new set $S^{'}_{T_o}(p_i) \subseteq S_{T_o}(p_i)$;
   2.3 find a t_j in $S^{'}_{T_o}(p_i)$ such that

$$RN(t_j) = \max_{\forall t' \in S^{'}_{T_o}(p_i)} (RN(t^{'}))$$

let $RN(t_j) = RN(t_j) - 1$.

3. Let

$$K_{min} = \max_{\forall t_i \in T_o} (RN(t_i))$$

◇

It is easy to check that the complexity of Algorithm 4.1 is $O(n^2)$, where n is the number of transitions of T_o.

Theorem 4.3 For any UPN-MTPN, $UPN - MTPN$, with an initial marking M_0, we can find K_{min} by Algorithm 4.1. K_{min} is the minimum number of registers such that M_0 is live for $UPN - MTPN$.

Proof: First we prove that M_0 with K_{min} registers is live for $UPN - MTPN$.

For any transition $t_i \in T_o$,
(1) if $\#(t_i, O(p_r)) + |I_{P_v}(t_i)| \leq K_{min}$, then at any time when we want to fire t_i, we can first fire some releasing or spilling transitions such that t_i is firable;
(2) if $\#(t_i, O(p_r)) + |I_{P_v}(t_i)| > K_{min}$, but $K_{min} \geq RN(t_i)$(see step 3 of Algorithm 4.1). Note that step 2.3 of Algorithm 4.1, so we can choose a special order of firing of transitions such that when we fire t_i, $K_{min} = \#(t_i, O(p_r)) + |I_{P_v}(t_i)| - the\ number\ of\ registers\ released\ by\ firing\ t_i$.

From (1) and (2), M_0 is live for $UPN - MTPN$.

Now we prove that K_{min} is the minimum number of registers. If the number of registers, K, is less than K_{min}, then there must be a $t_i \in T_o$, $RN(t_i) > K$. Since the number of tokens in $\{p_r\} \cup P_v$ doesn't change (Theorem 2.1 and 3.1), by any order of firing of transitions and at any time, we have $\#(t_i, O(p_r)) + |I_{P_v}(t_i)| - the\ number\ of\ registers\ which\ can\ be\ released\ by\ firing\ t_i > the\ number\ of\ tokens\ in\ \{p_r\} \cup P_v$. Therefore t_i cannot be fired.
□

Theorem 4.4 For a MTPN, $MTPN$, with an initial marking M_0, K_{min} is the minimum number of registers such that $MTPN$ with M_0 is schedulable.

Proof: We have to show that there exists a firable sequence, σ, of $UPN - MTPN$, such that $\bar{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_n)\ and\ \sigma_i = 1,\ i = 1, 2, ..., n,\ n = |T_o|$.

From Theorem 4.3, M_0 is live for $UPN - MTPN$. Note that (P_o, T_o, A_o) is a marked graph and the number of tokens in $\{p_r\} \cup P_v$ doesn't change and we can move tokens from places of P_v to p_r or from p_r to those of P_v by firing spilling transitions, so a new marking obtained by firing a transition of T_o from M_0 is still live.

Next, let us show that if $\bar{\sigma}$ is not the zero vector, then there exists at least one transition $t_i \in T_o$ such that $\sigma_i = 1$, which can be fired immediately or after firing some releasing or spilling transitions. Consider any t_i with $\sigma_i > 0$. If it is firable, the claim is proven. If not, construct a token-free path $t_j \rightarrow t' \rightarrow ... \rightarrow t'' \rightarrow t_i$ in (P_o, T_o, A_o) such that t_j is firable if we only consider (P_o, T_o, A_o). Since each place along this path has no token on it, so $\sigma_j = \sigma_{t'} = ... = \sigma_{t''} = \sigma_i = 1$. Now we can fire some spilling transitions or releasing transitions so that t_j can be fired immediately even if we consider the total $MTPN$. Let us fire t_j, and modify σ_j from 1 to 0. Repeat this process until $\bar{\sigma}$ is the zero vector.

Now, we have proven that there exists a firable sequence σ of $UPN - MTPN$ such that $\bar{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_n)$, where $\sigma_i = 1,\ i = 1, 2, ..., n,\ n = |T_o|$. By Theorem 4.1, $MTPN$ with M_0 is schedulable.

It is very obvious that K_{min} is the minimum number of registers since if the number of registers is less than K_{min}, then M_0 is not live for $UPN - MTPN$, thus there doesn't exist any firable sequence σ including all transitions of T_o.
□

5 Analysis of the MTPN and the Optimum Loop Schedule

In order to get the optimum loop schedule of a MTPN with an initial marking M_0, the common approach is to develop a state graph which can describe the behavior of the MTPN. However, a complete state graph includes all reachable markings from M_0. This means that the number of nodes in the state graph is exponential. Therefore, using this common approach to find the optimum loop schedule will suffer from exponential computation complexity.

In this section, we will present a new approach with polynomial computation complexity to find the optimum loop schedule for almost all practical loop programs. We first extend MTPN by introducing extended places which form a deadlock and trap, so the number of markings on these extended places is bounded by the number of tokens. Secondly, we develop a state subgraph, called Register Allocation Solution Graph (denoted as RASG), whose nodes consist of all reachable markings on the extended places. We will theoretically prove that RASG can effectively describe the major behavior of the MTPN, by which we can find the optimum loop schedule. We also prove that the number of all nodes in RASG is polynomial and our approach is of polynomial computation complexity.

5.1 Extending MTPN

We extend MTPN to EMTPN by introducing some extended places.

Definition 5.1 For the given MTPN, we can construct its EMTPN by the following steps.

1. For the register place p_r, build a new place p_{r-ext} (called extended register place); for each arc connected to p_r, copy this arc connected to p_{r-ext}.For each variable place $p_v(x)$, build a new place $p_{v-ext}(x)$ (called extended variable place); for each arc connected to $p_v(x)$, copy this arc connected to $p_{v-ext}(x)$. Let P_{rv-ext} represent the set of the extended register place and the extended variable places.

2. For the register place p_r, build a new place p_{r-sp} (called spilling register place); for each variable place $p_v(x)$, build a new place $p_{v-sp}(x)$ (called spilling variable place). For each pair of spilling transitions, $t_{ss}(x)$ and $t_{sl}(x)$, corresponding to $p_v(x)$, build four arcs, from p_{r-sp} to $t_{ss}(x)$, from $t_{ss}(x)$ to $p_{v-sp}(x)$, from $p_{v-sp}(x)$ to $t_{ls}(x)$ and from $t_{ls}(x)$ to p_{r-sp}, respectively. Let P_{sp-ext} represent the set of the spilling register place and spilling variable places.

3. Build a new place p_{rl-h} (called releasing head-place); for each operation transition $t(op)$ whose input includes variable place(s), build an arc from p_{rl-h} to $t(op)$; for each releasing transition $t_{rl}(x)$, build an arc from $t_{rl}(x)$ to p_{rl-h}. For each releasing place $p_{rl}(x, op)$, build a new place $p_{rl-ext}(x, op)$ (called extended releasing place); for each arc connected

to $p_{rl}(x, op)$, copy this arc connected to $p_{rl-ext}(x, op)$. Let P_{rl-ext} represent the set of the releasing head-place and extended releasing places.

4. The definition of weight on new arcs: For each new arc e from p_{rl-h} to $t(op)$, $W(e) =$ the number of the variable places which are the inputs of $t(op)$; for each new arc e from a releasing transition $t_{rl}(x)$ to p_{rl-h}, $W(e) =$ the number of the releasing places which are the inputs of $t_{rl}(x)$; For each new arc e from the extended register place p_{r-ext} to an operation transition $t(op)$, $W(e) = W((p_r, t(op)))$; for other new arc e, $W(e) = 1$.

5. The definition of time on new arcs: For each new arc e, $D(e) = 0$.

Fig.5.1 gives an example of EMTPN which is extended from the MTPN shown in Fig.3.1(f).

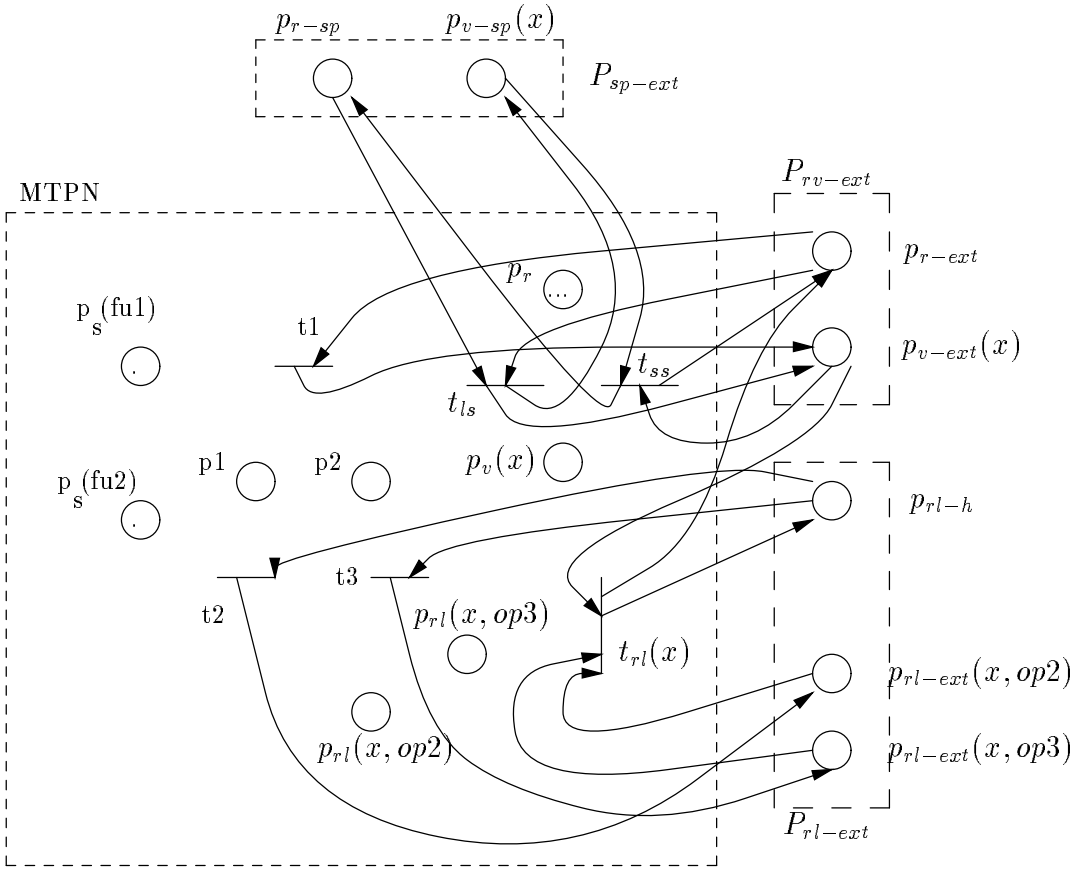


Fig.5.1 The EMTPN of the MTPN shown in Fig.3.1(f)

The initial marking, M_{0-ext}, of an EMTPN can be defined in terms of the initial marking M_0 of its corresponding MTPN.

Definition 5.2 Let M_{0-ext} be the initial marking of an EMTPN and M_0 be the initial marking of its corresponding MTPN; let $N_{op}(x)$ be the number of operations which reference variable x and N_{op}^{max} be $\max_{\forall x \in LDDG}(N_{op}(x))$, then

1. For any place p in MTPN, $M_{0-ext}(p) = M_0(p)$.

2. For the extended register place p_{r-ext}, $M_{0-ext}(p_{r-ext}) = M_0(p_r)$; for the spilling register place p_{r-sp}, $M_{0-ext}(p_{r-sp}) = M_0(p_r)$, where p_r is the register place.

3. For the releasing head-place p_{rl-h}, $M_{0-ext}(p_{rl-h}) = N_{op}^{max} * M_0(p_r)$.

4. For other place p, $M_{0-ext}(p) = 0$.

About EMTPN, there are two important properties which can be described by the following two theorems.

Theorem 5.1 Let M_0 be the initial marking of a MTPN and M_{0-ext} be the initial marking of its EMTPN. If M_0 is live for the MTPN, then M_{0-ext} is live for the EMTPN, too.

Proof: We can only consider the markings of the extended register place p_{r-ext}, the spilling register place p_{r-sp} and the releasing head-place p_{rl-h}.

The marking of p_{r-ext} doesn't make the operation transitions not firable since $M_{0-ext}(p_{r-ext}) =$ *the number of registers.*

The marking of p_{r-sp} doesn't make the load-spilling transitions not firable since $M_{0-ext}(p_{r-sp}) =$ *the number of registers.*

The marking of p_{rl-h} is $N_{op}^{max} * M_0(p_r)$, which is enough to fire any operation transition.
□

Theorem 5.2 In EMTPN, let $P_{ext} = P_{rv-ext} \cup P_{sp-ext} \cup P_{rl-ext}$, then P_{ext} is a deadlock&trap.

Proof: According to Definition 5.1, it is easy to check that P_{rv-ext}, P_{sp-ext} and P_{rl-ext} are a deadlock&trap respectively, so P_{ext} is a deadlock&trap.
□

5.2 Register Allocation Solution Graph

For the given computer architecture, the number of registers can be regarded as a constant by which we mean that it is independent of the number of operations in the loop programs. In this subsection, we will present a state subgraph of EMTPN, called Register Allocation Solution Graph (denoted as RASG), whose nodes only consist of all markings of the extended places (that is, P_{ext}). According to Definition 5.2 and Theorem 5.2, the number of nodes in RASG is bounded by the number of registers (that is why we name this state subgraph Register Allocation Solution Graph). We also theoretically point out that, for almost all practical programs, RASG can effectively describe the major behavior

of the MTPN by which we can find the optimum loop schedule in the next subsection.

Definition 5.3 In MTPN, for each $t_i \in T_{s-l}$, we call t_j the dual transition of t_i if $t_j \in T_{s-l},\ I(t_i) = O(t_j)\ and\ O(t_i) = I(t_j)$.

Definition 5.4 A Register-Allocation-Solution-Graph, $RASG = (V, E, TS, D_R)$, is a state subgraph of EMTPN with an initial marking M_{0-ext}, where

1. V is the set of nodes, in which each node corresponds to a reachable marking of P_{ext} from $M_{0-ext}(P_{ext})$, where $P_{ext} = P_{rv-ext} \cup P_{sp-ext} \cup P_{rl-ext}$;

2.E is the set of arcs;

3. TS is a function from E to the set of subsets of transitions;

4. D_R is a function from E to the set of nonnegative integers;

5. V, E, TS and D_R can be constructed as follows:

1) Let $\bar{V} = \{M_{0-ext}\},\ V = \{M_{0-ext}(P_{ext})\},\ E = \emptyset,\ M_{0-ext}(P_{ext})$ is called the initial marking node of $RASG$.

2) For each $M'_{ext} \in \bar{V}$, we choose the successors of $M'_{ext}(P_{ext})$ by the following rules:

Rule 5.1: At least one operation transition is fired between firing a spilling transition and its dual transition.

Rule 5.2: Releasing transitions must be fired as soon as they are firable.

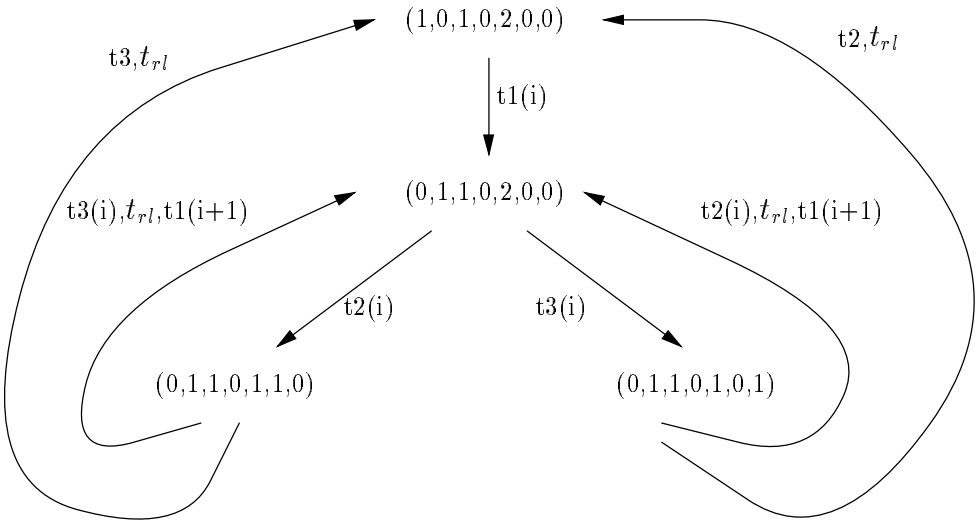
3) For each $M'_{ext} \in \bar{V}$, if ts is the set of simultaneously firable transitions which satisfies the above two rules, fire each transition of ts at the same time and generate a new marking M''_{ext}:
(1) $M''_{ext} \rightarrow \bar{V},\ M''_{ext}(P_{ext}) \rightarrow V$;
(2) Construct a new arc, e, from $M'_{ext}(P_{ext})$ to $M''_{ext}(P_{ext})$, $e \rightarrow E$;
(3) Let $TS(e) = ts$;
(4) Let $D_R(e) =$ The difference between the time when $M'_{ext}(P_{ext})$ was formed and the time when ts was firable (By Definition 5.1, for any ts_i, a set of simultaneously firable transitions, and the marking $M_{ext}(P_{ext})$ which is formed by firing ts_i, the time when $M_{ext}(P_{ext})$ is formed is in fact the time when ts_i is firable).

4) Repeat 2) and 3) until there is not any new node or arc to be constructed.

Figure 5.2 partly gives an example of the RASG corresponding to the EMTPN shown in Figure 5.1. In this example, we assume there is only one available register, t1 is addition operation with latency 1, t2 and t3 are store operations with latency 1. Obviously, the initial marking of P_{ext}, $(|p_{r-ext}|, |p_{v-ext}(x)|, |p_{r-sp}|, |p_{v-sp}(x)|, |p_{rl-h}|, |p_{rl-ext}(x, op2)|, |p_{rl-ext}(x, op3)|) = (1, 0, 1, 0, 2, 0, 0)$.

The following theorems give the important properties of RASG.

Theorem 5.3 In RASG, for all $v, v_i, v_j \in V$, if $(v, v_i), (v, v_j) \in E\ and\ v_i = v_j$, then $TS((v, v_i)) = TS((v, v_j))$.



(For any arc e, $D_R(e) = 1$. In $tj(i)$, i is the iteration number)

Fig.5.2 An example of RASG

Proof: Let $M_{ext}(P_{ext}) = v$, $M_{ext-i}(P_{ext}) = v_i$, $M_{ext-j}(P_{ext}) = v_j$, $ts_i = TS((v, v_i))$ and $ts_j = TS((v, v_j))$.

Assume $ts_i \neq ts_j$, then, from Rule 5.2, there may be only two cases as follows:

Case 1: There exists at least one spilling transition, $t_s^{'}$, such that $t_s^{'} \in ts_i$ *but* $t_s^{'} \notin ts_j$, thus there must exist one place $p^{'} \in P_{sp-ext}$, such that $M_{ext-i}(p^{'}) \neq M_{ext-j}(p^{'})$, that is $v_i \neq v_j$.

Case 2: There exists at least one operation transition, say $t^{'}$, such that $t^{'} \in ts_i$ *but* $t^{'} \notin ts_j$. After firing $t^{'}$, one token will be put into a place $p_{v-ext}^{'}$ of P_{v-ext} or (and) a place $p_{rl-ext}^{'}$ of P_{rl-ext}. There are also two cases:

Case 2.1: The token is put into $p_{v-ext}^{'}$. If there is not any spilling transition in ts_i taking the token away from $p_{v-ext}^{'}$, then $M_{ext-i}(p_{v-ext}^{'}) \neq M_{ext-j}(p_{v-ext}^{'})$; if yes, say $t_s^{'}$, then $t_s^{'} \in ts_j$, so $M_{ext-i}(p_{v-ext}^{'}) \neq M_{ext-j}(p_{v-ext}^{'})$.

Case2.2: If the token is not put into $p_{v-ext}^{'}$, then it must be put into $p_{rl-ext}^{'}$. If there is not any releasing transition in ts_i taking the token away from $p_{rl-ext}^{'}$, then $M_{ext-i}(p_{rl-ext}^{'}) \neq M_{ext-j}(p_{rl-ext}^{'})$; if yes, say $t_r^{'}$, if $t_r^{'} \in ts_j$, then $M_{ext-i}(p_{rl-ext}^{'}) \neq M_{ext-j}(p_{rl-ext}^{'})$; if $t_r^{'} \notin ts_j$, then there must exist a place $p_{v-ext}^{''}$ in P_{v-ext}, such that $M_{ext-i}(p_{v-ext}^{''}) \neq M_{ext-j}(p_{v-ext}^{''})$.

That is $v_i \neq v_j$.

Therefore, $ts_i = ts_j$.

□

Lemma 5.1 In any cycle of a RASG, any spilling transition and its dual transition are

fired the same number of times.

Proof: Let $v_1, v_2, ..., v_l$ be a cycle of $RASG$, $v_1 = v_l$, $M_{ext-1}(P_{ext}) = v_1$ and $M_{ext-l}(P_{ext}) = v_l$. We consider the firable sequence, $\sigma = TS((v_1, v_2))TS((v_2, v_3))... TS((v_{l-1}, v_l))$.

Let $t_i^s \in \sigma$ be a spilling transition and $\bar{t}_i^s$ its dual transition, then there is a place $p_i^{sp} \in P_{sp-ext}$, such that $I(p_i^{sp}) = t_i^s$ *and* $O(p_i^{sp}) = \bar{t}_i^s$ or $O(p_i^{sp}) = t_i^s$ *and* $I(p_i^{sp}) = \bar{t}_i^s$. Thus, if t_i^s and $\bar{t}_i^s$ are fired the different number of times in σ, then $M_{ext-1}(p_i^{sp}) \neq M_{ext-l}(p_i^{sp})$, that is $v_1 \neq v_l$. So t_i^s and $\bar{t}_i^s$ must be fired the same number of times in σ.
□

Theorem 5.4 For any LDDG of a loop, if there does not exist such a cut-set that it includes no variable defined-referenced edge, then in any cycle of the RASG corresponding to the LDDG, (1) all operation transitions are fired at least one time; (2) the operation transitions are fired the same number of times each.

Proof: (1) Let $C = v_1 v_2 ... v_l$ be a cycle of the RASG, $v_1 = v_l$. Let $M_{ext-1}(P_{ext}) = v_1$, $M_{ext-l}(P_{ext}) = v_l$, and $\sigma = TS((v_1, v_2))TS((v_2, v_3))...TS((v_{l-1}, v_l))$.

Assume that all operation transitions are not fired in σ, let $T_o^{'}$ be the set of operation transitions being fired in σ and $\bar{T}_o^{'}$ that not being fired in σ. By Rule 5.1, $T_o^{'} \neq \emptyset$. As there does not exist such a cut-set that it includes no variable defined-referenced edge in the LDDG, there may be only three cases as follows:

Case 1: There exist $t \in T_o^{'}$ and $\bar{t} \in \bar{T}_o^{'}$, such that there is a place $p_{v-ext} \in P_{v-ext}$ which $p_{v-ext} \in O(t) \cap I(\bar{t})$. Since $\bar{t}$ is not fired in σ and by Lemma 5.1 the spilling transitions in σ do not change the tokens of p_{v-ext}, so firing t will put a token into p_{v-ext} but the token won't be removed away, this means $M_{ext-1}(p_{v-ext}) \neq M_{ext-l}(p_{v-ext})$.

Case 2: There exist $t \in T_o^{'}$ and $\bar{t} \in \bar{T}_o^{'}$, such that there is a place $p_{v-ext} \in P_{v-ext}$ which $p_{v-ext} \in O(\bar{t}) \cap I(t)$ and there is not any other $\bar{t}^{'} \in \bar{T}_o^{'}$ such that $p_{v-ext} \in I(\bar{t}^{'}) \cap O(\bar{t})$. Since $\bar{t}$ is not fired in σ and by Lemma 5.1 the spilling transitions in σ do not change the tokens of p_{v-ext}, so firing t will remove a token away from p_{v-ext} but the token won't be put back into p_{v-ext}, this means $M_{ext-1}(p_{v-ext}) \neq M_{ext-l}(p_{v-ext})$.

Case 3: There exist $t \in T_o^{'}$ and $\bar{t}, \bar{t}^{'} \in \bar{T}_o^{'}$ such that there is a place $p_{v-ext} \in P_{v-ext}$ which $p_{v-ext} \in O(\bar{t}) \cap I(t) \cap I(\bar{t}^{'})$. then there must be $p, p^{'} \in P_{rl-ext}$ which $I(p) = t$ and $I(p^{'}) = \bar{t}^{'}$, and there is also a releasing transition t_{rl} which $t_{rl} = O(p) = O(p^{'})$. If t_{rl} is fired in σ, then firing t_{rl} will remove a token away from $p^{'}$ but the token won't be put back into $p^{'}$ since $\bar{t}^{'}$ is not fired in σ. This means $M_{ext-1}(p^{'}) \neq M_{ext-l}(p^{'})$. If t_{rl} is not fired in σ, then firing t will put a token into p but the token won't be remove away. This also means $M_{ext-1}(p) \neq M_{ext-l}(p)$.

Therefore, all operation transitions are fired in σ.

(2) For any operation transitions t_i, t_j which there is a place $p_{v-ext} \in P_{v-ext}$ such that $p_{v-ext} \in O(t_i) \cap I(t_j)$, let t_{rl} be a releasing transition which $p_{v-ext} \in I(t_{rl})$, so t_i and t_{rl} are

fired the same number of times in σ since there is only t_i which can put token into p_{v-ext} and there is only t_{rl} which can remove away token from p_{v-ext} besides spilling transitions and, notice that, spilling transitions in σ do not change the tokens of p_{v-ext} by Lemma 5.1.

Let p_{rl-ext} be a place of P_{rl-ext} which $I(p_{rl-ext}) = t_j$ and $O(p_{rl-ext}) = t_{rl}$, then t_j and t_{rl} are fired the same number of times in σ since there is only t_j which can put token into p_{rl-ext} and there is only t_{rl} which can remove away token from p_{rl-ext}.

Therefore, t_i and t_j are fired the same number of times in σ. That is, the operation transitions are fired the same number of times each.
□

Theorem 5.5 is an immediate consequence of Theorem 5.3 and Theorem 5.4.

Theorem 5.5 For a MTPN with an initial marking M_0 and its RASG, $(MTPN, M_0)$ and $RASG$, if there does not exist such a cut-set that it includes no variable defined-referenced edge in its corresponding LDDG, then a cycle of $RASG$ passing through the initial marking node corresponds to a N-schedule of $MTPN$ with M_0, N is the number of firing times of each operation transition in the cycle.

The following theorem gives the upper bound of the number of the nodes in RASG.

Theorem 5.6 The number of the nodes in RASG is at most $O(n^{N_r(N_{op}^{max}+2)})$, where n is the number of the operations in its corresponding $LDDG$, N_r the number of registers, $N_{op}(x)$ the number of operations referencing variable x and

$$N_{op}^{max} = \max_{\forall x \in LDDG}(N_{op}(x))$$

Proof: From Theorem 5.2, P_{ext} is a deadlock&trap. The number of the initial tokens in P_{ext} is $N_r + N_r + N_{op}^{max} \times N_r = N_r \times (N_{op}^{max} + 2)$. From Theorem 2.1, the number of the tokens in P_{ext} is not changed for any firing sequence, this means the total number of the markings in P_{ext} is $O(n^{N_r(N_{op}^{max}+2)})$, therefore, the number of the nodes in RASG is at most $O(n^{N_r(N_{op}^{max}+2)})$.
□

5.3 Finding the Optimum Loop Schedule

Using RASG as a basis, we can analyze the major behavior of MTPN and present a new loop scheduling approach. For almost all practical loop programs, our approach can find the optimum loop schedule under the resource constraints, with the polynomial computation complexity.

According to Definition 5.5 and the definition of MTPN, a N-schedule of MTPN with an initial marking M_0 corresponds to a loop schedule. By Theorem 5.5, a cycle of RASG

passing through $M_{0-ext}(P_{ext})$ corresponds to a N-schedule of MTPN with M_0. Thus, in order to find the optimum loop schedule, we can only focus on the strongly connected component $\overline{RASG}$ of RASG which includes the initial marking node $M_{0-ext}(P_{ext})$.

Definition 5.5 For any cycle C of $\overline{RASG}$, we define

$$\gamma(C) = N(C)/D_R(C)$$

where $N(C)$ is the number of firing times of each operation transition on C. and

$$D_R(C) = \sum_{\forall e \in C} D_R(e)$$

$\gamma(C)$ is called as the computation rate of the loop schedule corresponding to the cycle C.

Definition 5.6 For a RASG and its $\overline{RASG}$, we define

$$\gamma_{opt} = \max_{\forall C \in \overline{RASG}}(\gamma(C))$$

A critical cycle of $\overline{RASG}$ is a simple cycle C which $\gamma(C) = \gamma_{opt}$.

Note that, in the definition of RASG, Rule 5.1 only guarantees to eliminate all cycles which consist of spilling transitions, and Rule 5.2 only guarantees that all registers can come back the register place as soon as they are free. Therefore, the optimum loop schedule can be found in the cycles of $\overline{RASG}$.

Theorem 5.7 For the given loop program and computer architecture, we can construct LDDG, MTPN, EMTPN, RASG and $\overline{RASG}$, any critical cycle of $\overline{RASG}$ corresponds to an optimum loop schedule if there does not exist such a cut-set in the LDDG that it includes no variable defined-referenced edge.

Proof: Let C be a critical cycle of $\overline{RASG}$ and the number of firing times of each operation transition in C be N_0. If C passes through the initial marking node, C corresponds to a $m \times N_0$-schedule, m is an arbitrary positive integer. If C doesn't pass through the initial marking node (denoted as v_0), then there must be a node, v, in C which there exist a path from v_0 to v (denoted as $l_{v_0 \to v}$) and a path from v to v_0 (denoted as $l_{v \to v_0}$) since $\overline{RASG}$ is a strongly connected graph. Let $l_{v_0 \to v} l_{v \to v_0}$ corresponds to a k_0-schedule, then $l_{v_0 \to v} C^m l_{v \to v_0}$ correponds to a $(k_0 + m \times N_0)$-schedule.

Therefore, any critical cycle of $\overline{RASG}$, C, corresponds to a $(k_0 + m \times N_0)$-schedule, k_0 is some nonnegative integer and m an arbitrary positive integer.

For simplicity, assume the number of iterations of the loop, n, is large enough and can be expressed as $(k_0 + m \times N_0)$. An optimum loop schedule is a n-schedule which satisfies

$$\lim_{n \to \infty}(\gamma) = \gamma_{opt}.$$

Now, for $l_{v_0 \to v} C^m l_{v \to v_0}$, $n = k_0 + m \times N_0$,

$$\lim_{n \to \infty} m = \infty$$

thus

$$\lim_{n \to \infty} \gamma(l_{v_0 \to v} C^m l_{v \to v_0}) = \lim_{m \to \infty} \gamma(l_{v_0 \to v} C^m l_{v \to v_0}) = \gamma(C) = \gamma_{opt}$$

so $l_{v_0 \to v} C^m l_{v \to v_0}$ corresponds to an optimum loop schedule, that is C corresponds to an optimum loop schedule.
□

Theorem 5.7 in fact gives an approach to find the optimum loop schedule and the condition under which the approach can work. In the next section, we will further discuss this condition and point out that, for almost all practical loop program, this condition can be satisfied. Now we summary our approach as follows:

Algorithm 5.1

1. Build the LDDG of the given loop;
2. Build the MTPN;
3. Build the EMTPN;
4. Build the RASG;
5. Find the strongly connected component $\overline{RASG}$ passing through the initial marking node;
6. Find a critical cycle of $\overline{RASG}$, the sequence of the transitions along this cycle consists of the loop body of the optimum loop schedule;
7. Compute γ of this critical cycle which is the maximum computation rate of the given loop under the limitation of resources. ◇

Theorem 5.8 The computation complexity of Algorithm 5.1 is $O(n^{N_r(N_{op}^{max}+2)})$ in the worst case, where n is the number of operations, N_r the number of registers, and N_{op}^{max} is defined in Theorem 5.6.

Proof: The most time-expensive steps are step 5 and step 6. Finding the strongly connected component $\overline{RASG}$ is the complexity of $O(m^3)$, where m is the number of nodes of RASG. Finding a critical cycle of the $\overline{RASG}$ is the complexity of $O(\bar{m}^3)$, where $\bar{m}$ is the number of nodes of $\overline{RASG}$. Note that the number of nodes of RASG is $O(n^{N_r(N_{op}^{max}+2)})$ (by Theorem 5.6), thus the computation complexity of Algorithm 5.1 is $O(n^{3N_r(N_{op}^{max}+2)})$ in the worst case.
□

We believe that, in almost all practical loop programs, N_{op}^{max} is independent of n. In addition, let us consider the transformation shown in Figure 5.3. If the number of operations referencing variable x is greater than 2, we can transform the LDDG into a new one in which the number of operations referencing x is equal to 2. So the computation complexity of our approach is polynomial.

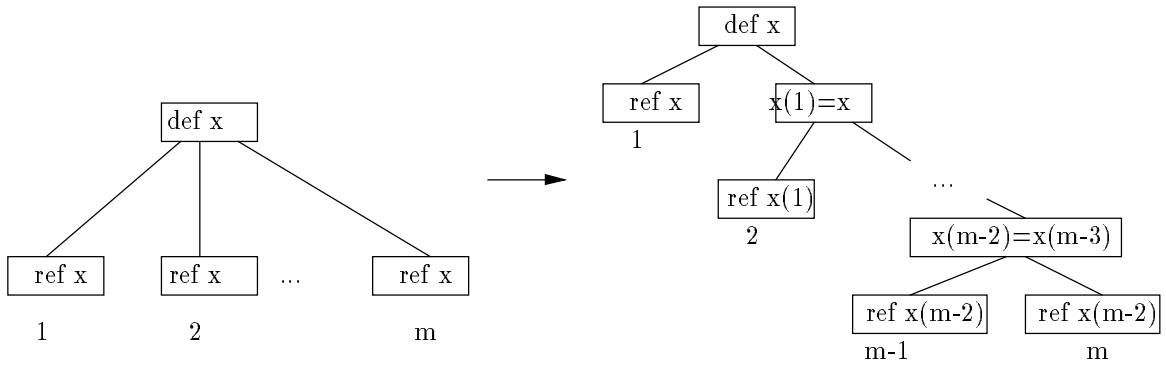


Fig.5.3 A transformation

6 Discussion

In this section, we first discuss the condition given in Theorem 5.5 and Theorem 5.7 under which our approach can work, and then discuss the applications of our model and approach.

The condition in fact requires that the given LDDG is connected by variable defined-referenced edges. That is , for any two subgraphs of the LDDG, there is at least a variable defined-referenced edge connecting them. We first demonstrate the ambiguity dependence edges which may exist in the LDDG. For example, op_i reads an array element, $A[exp1]$, and op_j writes an array element, $A[exp2]$. If we can not determine if $exp1$ is equal to $exp2$ in compiling time, then we must consider that there is a data dependence between op_i and op_j, which we call an ambiguity dependence. After intermediate code generation and variable renaming, the LDDG only includes variable defined-referenced edges and ambiguity dependence edges. Now we consider the following two cases.

For the first cases, the LDDG is not connected, we can break up the loop into several ones and handle each of these new loops respectively. Breaking up a loop program is sometimes very profitable, especially for loop vectorization.

For the second cases, the LDDG is connected. We can reduce the condition to a much weaker constraint, that is, in the LDDG, there does not exist such a cut-set that it only includes ambiguity dependence edges. We have analyzed the benchmarking loop

programs of Lawrence Livermore. In 24 loop-kernels, except for kernel 15, 16, 17, 20, 22 and 24 (these loops include conditional jumps, but our model and approach can only handle the loops without any conditional jump. Our future work is to extend our model and approach to handle the loops with conditional jumps), the other 18 kernels satisfy the above constraint. We believe that, for almost all practical loop programs, the above constraint is often satisfied.

We expect that our model and approach can be applied to the design of high performance computer architectures as well as the design of optimizing compilers. For the design of high performance computer architectures, especially for the application-specific high performance computers, it is required that the architectures should match very well the loop programs to be executed on them, such that these loop programs can be executed at the maximum computation rates. Our model and approach can effectively analyze the maximum computation rates of loop programs executed on the given architectures, so they can provide very useful information for selection of the number of functional units and the number of registers. On the other hand, for the design of optimizing compilers, our model and approach are very effective tools to analyze the effects of high-level program transformations on the instruction-level parallelism for loop programs. In addition, our model and approach can be directly used in the register allocation in which they can effectively estimate the number of registers that will be allocated to the local variables in the most inner loops.

7 Conclusion

Under resource constraints, finding the optimum loop schedule for any loop program has been proven to be an NP complete problem. In this report, we present a new timed Petri net model to integrate loop scheduling, functional unit allocation, register allocation and spilling into a unified theoretical framework, and theoretically discuss the schedulability of this model. An efficient analysis tool, Register Allocation Solution Graph (RASG), is defined and constructed. Using RASG as a basis, we theoretically prove that, under resource constraints, finding the optimum loop schedules for almost all practical loop programs can be solved with polynomial computation complexity. An optimum loop scheduling approach with polynomial computation complexity is also given.

However, the computation complexity of our approach in the worst case is still great since the number of registers is expressed in exponential terms, in result, our approach can not be directly used as a loop scheduling approach in the practical optimizing compilers. Our future work is to develop some useful heuristics that efficiently reduce the number of nodes in RASG and to make our approach more practical. We will also extend our model and approach to handle the loop programs with conditional jumps.

References

[Aik87] A. Aiken and A. Nicolau, Perfect Pipelining: A New Loop Parallelization Technique, Research Report 87-873, Dept. of Computer Science, Cornell Univ., 1987.

[Alm89] G.S. Almasi and A. Gottlieb, Highly parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1989.

[Bod89] F. Bodin, et al., Overview of a High-Performance Programmable Pipeline Architecture, Proc. of ACM International Conference on Supercomputing, Crete, 1989.

[Bra91] D.G. Bradlee, et al., Integrating Register Allocation and Instruction Scheduling for RISCs, Proc. Intl. Conf. ASPLOS, April 1991.

[Car84] J. Carlier, et al., Modelling Scheduling Problems with Timed Petri Nets, Advances in Petri Nets 1984.

[Car88] J.Carlier, et al., Timed Petri Net Schedules, Advances in Petri Nets 1988.

[Cha81] A.E. Charlesworth, An Approach to Scientific Array Processing: The Architecture Design of the AP-120B/FPS-164 Family, Computer,9(1981), pp.18-27.

[Com71] F. Commoner and A.W. Holt, Marked Directed Graphs, Journal of Computer and System Sciences, 5:511-523, 1971.

[Eis88] C. Eisenbeis, Optimization of Horizontal Microcode Generation for Loop Structures, Proc. of ACM International Conference on Supercomputing, Saint-Malo, France, 1988.

[Eis89] C. Eisenbeis, et al., Compile-Time Optimization of Memory Usage on the CRAY-2, Proc. NATO Supercomputer Workshop, Trondheim, Norway, 1989.

[Gao91] G. R. Gao, et al., A Timed Petri-Net Model for Fine-Grain Loop Scheduling, Proc. of ACM SIGPLAN'91 Conference on PLDI, June, 1991.

[Gao92] Q. Ning and G.R. Gao, Optimal Memory Allocation for Argument Fetching Dataflow Machines, ACAPS Technical Memo 32, Mac Gill University, Montreal, Canada.

[Han87] C. Hanen, Problèmes D'Ordonnancement des Architectures Pipelines: Modélisation, Optimisation, Algorithmes, Thèse de Doctorat de L'Université Paris 6, 1987.

[Han91] C. Hanen, Study of a NP-Hard Cyclic Scheduling Problem: The Periodic Recurrent Job-Shop, Research Rapport, Laboratoire MASI, Universite P. et M. Curie, 1991.

[Lam88] M.S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machine. Proc. SIGPLAN'88 Conference on PLDI, Atlanta, June, 1988.

[Pet81] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice- Hall, Inc., Englewood Cliffs, NJ, 1981.

[Rei85] W. Reisig, Petri Nets: An Introduction, Springer-Verlag, Berlin Heidelberg, 1985.

[Su91] Bogong Su and Jian Wang, Loop-carried Dependence and the General URPR Software Pipelining Approach, Proc. 24th HAWAII International Conference on System Sciences, Kailua-Kona, Hawaii, Jan. 1991.

[Tou84] R.F. Touzeau, A Fortran Compiler for the FPS-164 Scientific Computer, Proc. ACM SIGPLAN Symposium on Compiler Construction, 1984.